Конспект лекции
Модульное тестирование на базе JUnit5. Основы тестирования

Цель и задачи лекции

Цель – научиться писать модульные тесты на базе JUnit5.

Задачи:

1. Ознакомиться с понятием модульного тестирования
2. Узнать, как использовать JUnit5 и Mockito

План занятия

1. Mockito

Mockito

Назначение фреймворка

Фреймворк Mockito предоставляет ряд возможностей для создания заглушек вместо реальных классов или интерфейсов при написании JUnit тестов.

Подключение зависимостей

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version> 2.23.0</version>
    <scope>test</scope>
</dependency>
```

Синтаксис создания заглушки Mockito

JUnit и фреймворк Mockito

Разработчикам программного обеспечения на разных этапах своей деятельности приходится сталкиваться с тремя стратегиями тестирования : функциональным, интеграционным и модульным тестированием. Все они используются для тестирования приложений разными способами, и каждая из стратегий имеет определенную цель. К сожалению, ни одна из стратегий не даёт стопроцентной гарантии обнаружения всех имеющихся ошибок. И даже комбинация всех трёх стратегий не может дать такой гарантии. Но их сочетание позволяет существенно снизить количество ошибок и убедить разработчика, что приложение функционирует согласно предъявленным требованиям.

Функциональное тестирование

Проведение функционального тестирования, как правило, связано с созданием специальной группы специалистов, занимающихся тестированием. На этом этапе приложение развертывается в действующем окружении и проверяется его соответствие ТЗ (Техническому Заданию) и предъявленным функциональным требованиям. Команда тестировщиков использует комплекс автоматизированных и ручных тестов.

Автоматизировать процесс функционального тестирования можно, если приложение включает API (Application Programming Interface) - интерфейс

прикладного программирования, на котором оно построено. Однако наличие интерфейса в приложении (desktop, web) существенно снижает возможности полной автоматизации данного процесса.

Интеграционное тестирование

Стратегия интеграционного тестирования основывается на проверке прикладного кода в окружении, близком к фактическому окружению, но не являющимся им. Главная цель данной стратегии – убедиться в правильности взаимодействия кода с внешними ресурсами и взаимодействия различных технологий в приложении между собой.

В интеграционном тестировании не требуется использовать фиктивные данные, как при модульном тестировании. Вместо этого в интеграционных тестах часто используются базы данных, находящиеся в памяти, которые легко можно создавать и уничтожать во время выполнения тестов. База данных в памяти – это самая настоящая база данных, что дает возможность проверить правильность работы сущностей JPA. Но все же эта база данных не совсем настоящая – она лишь имитирует настоящую базу данных для целей интеграционного тестирования.

Модульное тестирование

Целью модульного тестирования является проверка работы прикладной логики всего приложения или отдельных его частей при разных исходных данных, и анализ правильности получаемых результатов. Несмотря на то, что цель модульного тестирования выглядит простой и понятной, реализация этого типа тестирования может оказаться очень сложным и запутанным делом, особенно при наличии «старого» кода. Основные приемы проведения модульного тестирования опираются на следующие базовые принципы :

- внешние ресурсы не используются, т.е. недопустимо подключение к базам данных, веб-службам и т.п.;
- каждый класс имеет свой тест;
- тестируются только общедоступные методы или интерфейсы, а внутренний код тестируется за счет изменения входных данных;
- для получения данных, требуемых тестируемой логике, должны создаваться фиктивные зависимости.

При проведении модульного тестирования для создания фиктивных классов-зависимостей можно использовать простой, но мощный фреймворк Mockito совместно с JUnit.

Фреймворк Mockito

Фреймворк Mockito предоставляет ряд возможностей для создания заглушек вместо реальных классов или интерфейсов при написании JUnit

тестов. Mockito можно скачать с сайта https://code.google.com/p/mockito, либо определить в зависимостях (dependencies) в maven проекте :

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
</dependency>
```

Наибольшее распространение получили следующие возможности Mockito :

- создание заглушек для классов и интерфейсов;
- проверка вызова метода и значений передаваемых методу параметров;
- использование концепции «частичной заглушки», при которой заглушка создается на класс с определением поведения, требуемое для некоторых методов класса;
- подключение к реальному классу «шпиона» spy для контроля вызова методов.

Синтаксис создания заглушки Mockito

Чтобы создать Mockito объект можно использовать либо аннотацию @Mock, либо метод mock. В следующем примере в двух разных классах (Test_Mockito1, Test_Mockito2) разными способами создаются объекты mcalc для имитации интерфейса калькулятора ICalculator.

```
import org.mockito.Mock;
import org.mockito.Mockito;

import com.example ICalculator;

public class Test_Mockito1
{
    @Mock
    ICalculator mcalc;
}


public class Test_Mockito2
{
    ICalculator mcalc = Mockito.mock(ICalculator.class);
}
```

Если использовать статический импорт Mockito, то синтаксис будет иметь следующий вид :

```
import static org.mockito.Mockito.*;

import com.example ICalculator;
```

```
public class Test_Mockito
{
    ICalculator mcalc = mock(ICalculator.class);
}
```

Помните, что методы mock объекта возвращают значения по умолчанию : false для boolean, 0 для int, пустые коллекции, null для остальных объектов.

Методы Mockito в примерах

Для рассмотрения методов фреймворка Mockito будем использовать в качестве тестового класса калькулятор, реализующий интерфейс ICalculator. В следующем коде представлены листинги интерфейса ICalculator и класса Calculator.

Листинги тестового класса и интерфейса

```
public interface ICalculator
{
    public double add      (double d1, double d2);
    public double subtract (double d1, double d2);
    public double multiply (double d1, double d2);
    public double divide   (double d1, double d2);
}
//-------------------------------------------------
public class Calculator
{
    ICalculator icalc;

    public Calculator(ICalculator icalc) {
        this.icalc = icalc;
    }
    public double add(double d1, double d2) {
        return icalc.add(d1, d2);
    }
    public double subtract(double d1, double d2) {
        return icalc.subtract(d1, d2);
    }
    public double multiply(double d1, double d2) {
        return icalc.multiply(d1, d2);
    }
    public double divide(double d1, double d2) {
        return icalc.divide(d1, d2);
    }
    public double double15() {
        return 15.0;
    }
}
```

Как видно из листингов все методы калькулятора (add, subtract, multiply, divide), за исключением одного, возвращают не вычисленные значения, а результаты выполнения данных методов в объекте, реализующим интерфейс ICalculator, который в наших тестах будет представлять заглушка mcalc. Последний метод double15() должен вернуть реальное значение.

1. Определение поведения - when(mock).thenReturn(value)

Этот метод позволяет определить возвращаемое значение при вызове метода mock с заданными параметрами. Если будет указано более одного возвращаемого значения, то они будут возвращены методом последовательно, пока не вернётся последнее; после этого при последующих вызовах будет возвращаться только последнее значение. Таким образом, чтобы метод всегда возвращал одно и то же значение, следует просто определить одно условие.

```
@RunWith(MockitoJUnitRunner.class)
public class Test_Mockito
{
    @Mock
    ICalculator mcalc;

    // используем аанотацию @InjectMocks для создания mock объекта
    @InjectMocks
    Calculator calc = new Calculator(mcalc);

    @Test
    public void testCalcAdd()
    {
        // определяем поведение калькулятора для операции сложения
        when(calc.add(10.0, 20.0)).thenReturn(30.0);

        // проверяем действие сложения
        assertEquals(calc.add(10, 20), 30.0, 0);
        // проверяем выполнение действия
        verify(mcalc).add(10.0, 20.0);

        // определение поведения с использованием doReturn
        doReturn(15.0).when(mcalc).add(10.0, 5.0);

        // проверяем действие сложения
        assertEquals(calc.add(10.0, 5.0), 15.0, 0);
        verify(mcalc).add(10.0, 5.0);
    }
}
```

Метод verify позволяет проверить, была ли выполнена проверка с определенными параметрами. Если проверка не выполнялась или выполнялась с другими параметрами, то verify вызовет исключение.

Для определения поведения mock в тесте была использована также следующая конструкция:

```
doReturn(value).when(mock).method(params)
```

2. Подсчет количества вызовов - atLeast, atLeastOnce, atMost, times, never

Для проверки количества вызовов определенных методов Mockito предоставляет следующие методы:

- atLeast (int min) - не меньше min вызовов;
- atLeastOnce () - хотя бы один вызов;
- atMost (int max) - не более max вызовов;
- times (int cnt) - cnt вызовов;
- never () - вызовов не было;

Следующий тест демонстрирует контроль количества вызовов метода subtract с разными параметрами. Для этого сначала определяется поведение mock (при определенных параметрах выдавать соответствующие результаты), и выполняются проверки с использованием assertEquals. После этого выполняется проверка количества вызовов mock с определенными параметрами. Две последние проверки не выполняются - «закомментированы». Если снять комментарий хотя бы с одной из них, то метод verify, вызовет исключение. Комментарий к этим проверкам описывает причину вызова методом исключения.

```
@Test
public void testCallMethod()
{
    // определяем поведение (результаты)
    when(mcalc.subtract(15.0, 25.0)).thenReturn(10.0);
    when(mcalc.subtract(35.0, 25.0)).thenReturn(-10.0);

    // вызов метода subtract и проверка результата
    assertEquals (calc.subtract(15.0, 25.0), 10, 0);
    assertEquals (calc.subtract(15.0, 25.0), 10, 0);

    assertEquals (calc.subtract(35.0, 25.0),-10, 0);

    // проверка вызова методов
    verify(mcalc, atLeastOnce()).subtract(35.0, 25.0);
    verify(mcalc, atLeast   (2)).subtract(15.0, 25.0);

    // проверка - был ли метод вызван 2 раза?
    verify(mcalc, times(2)).subtract(15.0, 25.0);
    // проверка - метод не был вызван ни разу
    verify(mcalc, never()).divide(10.0,20.0);

    /* Если снять комментарий со следующей проверки, то
     * ожидается exception, поскольку метод "subtract"
     * с параметрами (35.0, 25.0) был вызван 1 раз
     */
    // verify(mcalc, atLeast (2)).subtract(35.0, 25.0);

    /* Если снять комментарий со следующей проверки, то
     * ожидается exception, поскольку метод "subtract"
     * с параметрами (15.0, 25.0) был вызван 2 раза, а
     * ожидался всего один вызов
     */
    // verify(mcalc, atMost (1)).subtract(15.0, 25.0);
}
```

3. Обработка исключений - when(mock).thenThrow()

Mockito позволяет вызвать исключение при определенных условиях. Для этого необходимо использовать следующий синтаксис кода:

```
// создаем исключение
RuntimeException exception = new RuntimeException ("Division by zero");
// определение поведения для вызова исключения
doThrow(exception).when(mock).divide(5.0, 0));
```

В представленном коде создали исключение RuntimeException. После этого определили условия вызова исключения - вызов метода деления на 0. В следующем тесте выполняется проверка метода divide. При делении на 0 вызывается исключение.

```
@Test
public void testDevide()
{
    when(mcalc.divide(15.0, 3)).thenReturn(5.0);

    assertEquals(calc.divide(15.0, 3), 5.0, 0);
    // проверка вызова метода
    verify(mcalc).divide(15.0, 3);

    // создаем исключение
    RuntimeException exception = new RuntimeException ("Division by zero");
    // определяем поведение
    doThrow(exception).when(mcalc).divide(15.0, 0);

    assertEquals(calc.divide(15.0, 0), 0.0, 0);
    verify(mcalc).divide(15.0, 0);
}
```

4. Использование интерфейса org.mockito.stubbing.Answer<T>

Иногда описание поведения mock объекта требует определенной проверки с усложнением логики. В этом случае можно использовать интерфейс Answer<T>, который позволяет реализовать заглушки методов со сложным поведением. В следующем тесте testThenAnswer при вызове метода сложения с определенными параметрами mcalc.add(11.0, 12.0) будет вызван метод answer, который подготовит ответ. Параметр InvocationOnMock позволяет получить информацию о вызываемом методе и параметрах.

```
// метод обработки ответа
private Answer<Double> answer = new Answer<Double>() {
    @Override
    public Double answer(InvocationOnMock invocation) throws Throwable
    {
        // получение объекта mock
        Object mock = invocation.getMock();
        System.out.println ("mock object : " + mock.toString());

        // аргументы метода, переданные mock
        Object[] args = invocation.getArguments();
        double d1 = (double) args[0];
        double d2 = (double) args[1];
        double d3 = d1 + d2;
        System.out.println ("" + d1 + " + " + d2);

        return d3;
    }
};
@Test
public void testThenAnswer()
{
    // определение поведения mock для метода с параметрами
    when(mcalc.add(11.0, 12.0)).thenAnswer(answer);
    assertEquals(calc.add(11.0,12.0), 23.0, 0);
}
```

5. Использование шпиона spy на реальных объектах

Mockito позволяет подключать к реальным объектам «шпиона» spy, контролировать возвращаемые методами значения и отслеживать количество вызовов методов. В следующем тесте создадим шпиона scalc, который подключим к реальному калькулятору и будем вызывать метод double15(). Необходимо отметить, что метод реального объекта double15 должен вернуть значение 15. Однако Mockito позволяет переопределить значение и согласно вновь назначенному условию новое значение должно быть 23.

```
@Test
public void testSpy()
{
    Calculator scalc = spy(new Calculator());
    when(scalc.double15()).thenReturn(23.0);

    // вызов метода реального класса
    scalc.double15();
    // проверка вызова метода
    verify(scalc).double15();

    // проверка возвращаемого методом значения
    assertEquals(23.0, scalc.double15(), 0);
    // проверка вызова метода не менее 2-х раз
    verify(scalc, atLeast(2)).double15();
}
```

В результате выполнения теста видим, что метод возвращает значение 23. Таким образом, фреймворк Mockito в сочетании с JUnit можно использовать для тестов реального класса. При этом, можно проверить, вызывался ли метод, сколько раз и с какими параметрами. Кроме этого, можно создавать заглушки только для некоторых методов. Это позволяет проверить поведение одних методов, используя заглушки для других.

6. Проверка вызова метода с задержкой, timeout

Фреймворк Mockito позволяет выполнить проверку вызова определенного метода в течение заданного в timeout времени. Задержка времени определяется в милисекундах.

```
@Test
public void testTimout()
{
    // определение результирующего значения mock для метода
    when(mcalc.add(11.0, 12.0)).thenReturn(23.0);
    // проверка значения
    assertEquals(calc.add(11.0,12.0), 23.0, 0);

    // проверка вызова метода в течение 10 мс
    verify(mcalc, timeout(100)).add(11.0, 12.0);
}
```

7. Использование в тестах java классов

В следующем тесте при создании mock объектов используются java классы Iterator и Comparable. После этого определяются условия проверок и выполняются тесты.

```
@Test
public void testJavaClasses()
{
    // создание объекта mock
    Iterator<String> mis = mock(Iterator.class);
    // формирование ответов
    when(mis.next()).thenReturn("Привет").thenReturn("Mockito");
    // формируем строку из ответов
    String result = mis.next() + ", " + mis.next();
    // проверяем
    assertEquals("Привет, Mockito", result);

    Comparable<String> mcs = mock(Comparable.class);
    when(mcs.compareTo("Mockito")).thenReturn(1);
    assertEquals(1, mcs.compareTo("Mockito"));

    Comparable<Integer> mci = mock(Comparable.class);
    when(mci.compareTo(anyInt())).thenReturn(1);
    assertEquals(1, mci.compareTo(5));
}
```

Литература и ссылки

1. https://www.intuit.ru/studies/courses/1040/209/lecture/5409
2. https://junit.org/junit5/docs/current/user-guide/
3. https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html
4. https://www.baeldung.com/junit-5
5. https://www.baeldung.com/junit-5-preview
6. https://www.baeldung.com/mockito-junit-5-extension
7. https://igorski.co/java/junit/mockito-with-junit5/
8. http://java-online.ru/junit-mockito.xhtml

Вопросы для самоконтроля

1. Для чего нужно модульное тестирование?
2. Как создать модульный тест?
3. Какие основные методы утверждений вы помните (Assertions)?
4. Назовите основные аннотации, сопровождающие класс модульных тестов?
5. Что такое mock-объекты? В каких ситуациях удобно их использовать?
6. Чем отличается mock от spy?